Intelligent Motor Controller

by

Matthew Sorensen and Ben Weiss

A report submitted in partial fulfillment

of the requirements for

ME 498/599

University of Washington, Winter 2014

ABSTRACT

In the Intelligent Motor Control project, one of the most popular 3D printer motherboards, the RepRap Arduino Mega Pololu Shield (RAMPS), is extended to drive and synchronize a network of discrete and motor controllers, each of which has its own processor. In this way, we overcome several bottlenecks of the current single-processor solutions available, including the ability to drive additional motors, to use more complex motor drivers, and to enable closed loop motor control. The Intelligent Motor Control design was implemented in breadboard form and was shown to synchronously control a 3D printer's motion.

TABLE OF CONTENTS

TABLE OF FIGURES

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support and encouragement of the Solheim 3D Printing Lab at the University of Washington, which lent expertise, space, and equipment to the team over the course of the project, and to the Washington Open Object Fabricators group for their encouragement. The RAMPS and Grbl open source communities are also acknowledged for their outstanding contributions to the open source movement, and for laying the groundwork for what is presented below.

Additionally, Ben wishes to thank and acknowledge his wife, Katrina, who put up with far too much nerdy talk as a result of this project, and God, without Whom none of this would be possible.

INTRODUCTION

From its earliest days, the computational demands of 3D printing have been integrally connected with progress in the field. Modelling, processing, and slicing tasks have grown increasingly more powerful as personal computers have progressed in performance, and the decreasing costs of embedded hardware have in large part enabled the precipitous drop in printer prices over the last 10 years.

Fifteen years ago, any 3D printer available had a full-blown computer under the hood, while now, a single Arduino is sufficient to control a 5 axis FFF robot at a fraction of the cost. As the low-end printers have evolved, we have seen a trend from several processors (as with the Makerbot Thing-O-Matic), connected with an RS485 bus, to a single device which issues step/direction commands to the dedicated stepper motor driver chips. Today, the open-source 3D printing community relies heavily on an Arduino-based platform called RepRap Arduino Mega Pololu Shield (RAMPS) [1].

Although cost savings have been realized by switching to a single processor, the tradeoff has been a reduction in the quality of the processor's timing. Because the ATMega microcontroller (MCU) at the core of most low-end 3D printers today is tasked with simultaneously managing up to three heaters, five motors, an LCD and keypad, an SD card, and USB communications with the computer, in addition to interpreting G-Code and doing path planning, it must switch between tasks very quickly. Even though the AVR architecture allows some degree of prioritization and multitasking with interrupts, this great number of tasks, many of which are interrupt-driven, causes the processor to get bogged down. Interrupts get stacked up in a queue instead of executing immediately when they should, and the result is a subtle breakdown of the "hard real time" nature of the platform.

This effect is most prominently seen in the motor driver. Of the three RAMPS firmwares tested (Marlin, Sprinter, and Teacup), all three showed order-of-magnitude variations in the time

between “step” commands sent to the motor. This effect is not noticeable at medium speeds, because the motor’s inertia smooths out the rapid jerks caused by the driver, but at low speeds the jerk is clearly visible. It is also hypothesized (though not yet tested) that this jitter in the pulse frequency causes resonance effects to be seen at lower velocities.

In addition, it is increasingly desirable to do more complex motor driver and closed loop control computations in order to produce better-performing 3D printers. The additional computational burden of reading encoders and computing control laws exceeds the current capabilities of the ATMega series of MCUs.

At the moment, the most flexible electronics platform available supports up to five axes of open-loop step-direction motion. Although RAMPS is modular enough to support a wide range of stepper-drivers, it is not compatible with more exotic configurations such as clockwise/CCW interface drivers, DC motors, or four-axis positioning stages. Thus, an electronics platform offering a seamless upgrade path from conventional 3D-printer configurations to applications demanding more axes, higher-power drivers, or high-speed motion control is desirable. Splitting the motor controllers away from the motherboard also allows independent supply of power to each motor, a major advantage in some larger printers.

Perhaps, future generations of ever-faster microprocessors will make some of the above points into non-issues. For now, however, processor performance remains a significant bottleneck in the 3D printer performance growth curve, one which we have addressed in this project.

In the broader motor control world, the pyGestalt [3] project should be mentioned. It divides the motor control problem among a network of modules, each of which has its own MCU and controls its own module. A software stack on the host PC breaks the move commands up into packets for each motor module, handling synchronization using special packets. A formal design document makes the project extensible to different kinds of actuators and sensors, and coordinated motion along multiple axes is at least theoretically possible. This solution was not

used in our project, however, because the source code was not available, and making the PC interface compatible with current 3D printer frontends such as Pronterface would have proven difficult.

PROBLEM STATEMENT

One way to address these issues is a distributed motion-control architecture, with each axis executing on a dedicated processor, all of which communicate over a serial bus. In order for such an architecture to achieve wide adoption in the RepRap community, we identified additional constraints for the system:

- The new system must be compatible with existing toolpath generation and printer interface software. As the RepRap world is poorly standardized, such compatibility realistically means choosing between emulating all of the quirks of an existing firmware, and modifying an existing firmware to interface with the new architecture.
- The new system should utilize the existing hardware as much as possible. Although the introduction of a number of new processors obviously requires the addition of new hardware, the existing hardware implements functionality such as extruder temperature regulation that can be retained in the new system.
- Finally, the overall cost should be as low as possible. Adding new processors will doubtless increase overall project costs, but this effect should be minimized.

SOLUTION DEVELOPED

Based on these project constraints, we developed a prototype system in which the master node runs on an Arduino Mega2560, connected to a RAMPS 1.2 shield - a very representative example of traditional RepRap electronics. Any one of a number of equivalent compatible AVR-based systems could be substituted, as long as they expose an I2C bus and at least one spare GPIO pin. As the master node runs a standard (if heavily modified) firmware on

a standard hardware platform, all printer functionality not replaced by our system, such as heater control, behaves as normal. Each axis runs on a dedicated Teensy 3.1 [2], a $20 ARM development board with a significantly more powerful processor than competitively priced AVR systems. Although the extra computational power is not strictly required to implement this system, the Teensy's MK20DX256 processor's 72MHz clock speed, dedicated hardware for interfacing with quadrature encoders, and two-channel ADC were appealing in light of future work involving closed-loop control. In our implementation, the Teensy 3.1 interfaces with a Pololu A4988 bipolar stepper driver breakout board, chosen due to its ubiquity in the RepRap world. The master node is connected to the axis controllers with a two-wire I²C bus, as well as a single Sync line (see Figure 1).

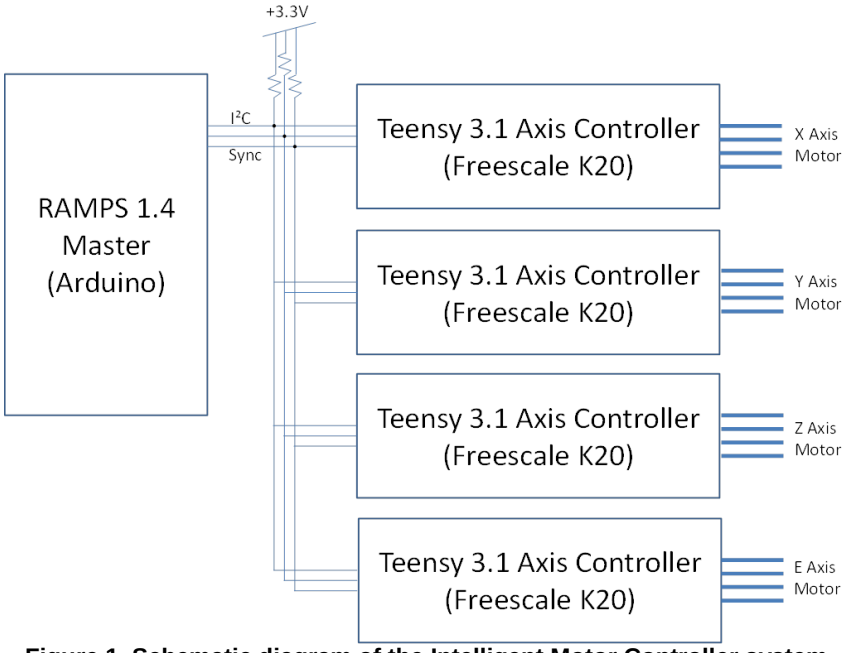

Figure 1. Schematic diagram of the Intelligent Motor Controller system.

Communication between nodes takes place over the 100kbps I2C line, with a custom packet-oriented binary protocol. Although the communication protocol is more-or-less OSI layer-1 agnostic and could use SPI, CAN bus, or equivalent, there are a number of compelling reasons for the choice of I2C. In particular, I2C simplifies addressing particular axes, as there is no need for a protocol-level or physical addressing mechanism. Additionally, hardware support

for asynchronous reading/writing of serial bytes is widely available in microcontrollers. The communication protocol consists of master-to-slave transmissions starting with a message type flag byte, followed by a payload of a fixed length for a given message type, and concluded with a checksum byte. Responses, which always immediately follow transmissions, start with a status byte, followed by a payload (fixed length for a given initial message type), and finish with a checksum byte. Different message types allow for queuing moves, querying axis status and position, and reading and writing axis parameters such as microstepping resolution. For a more detailed description of the communication protocol, consult Appendix A.

The master node, consisting of a RAMPS 1.2 shield over an Arduino Mega, runs a heavily modified version of the popular Marlin firmware. The changes primarily consist of rerouting all calls to the integrated stepper driver routines to the serial interface for distribution to the axis processors. Since the movement information transmitted to each slave is already present, almost verbatim, in the Marlin planner, no new processing of the move contents is required. Additionally, a series of new M-codes was introduced for manually interacting with the slaves, used mostly for debugging. In all, the changes to Marlin included substantial modifications of several files, as well as roughly 1000 lines of new C code to manage the serial interface.

The firmware running on each axis also consists of around 1000 lines of C, built with a heavily modified version of the Teensy 3.1 environment. Internally, it relies on a rewritten version of the Grbl trapezoidal velocity profile generator, originally ported to the platform as part of an ARM K20 port of the entire Grbl system [4]. Unlike Grbl, the slave's motor control algorithm has no need for features such as feed-holds and has a simpler approach to error recovery, but must instead integrate a cross-axis synchronization mechanism. Importantly, each slave runs exactly the same binary image - all configuration required to adapt a slave for a particular physical axis (motor resolution, homing direction, limit logic, etc.) takes place over the serial protocol, instead of the more tradition (in the RepRap world) compile-time configuration.

The system design takes advantage of the Teensy 3.1's abundance of IO - each slave picks up its address on the bus using 4 GPIO (allowing for up to 16 nodes).

In order to ensure coordinated linear motion, we developed a cross-axis synchronization mechanism independent of our serial communication bus. This is implemented by a shared synchronization line connected to a GPIO pin on every processor on the bus, and tied to +3.3V with one 4.7kOhm resistor. The execution of a move is triggered by a rising edge of the sync line, after which each device on the bus waits for a fixed interval (nominally, 10 us) to ensure complete signal propagation, and then drives their sync pin to ground to indicate that a move is in progress. When a device is done executing a move, it configures its sync pin as a high-impedance input, signaling move completion, and starts a timer. When every device on the bus has signaled completion, the line's pull-up resistor pulls the sync line to 3.3V, generating the rising edge that triggers the next move (see Figure 2). As the slaves each start timers when they finish a move, each device tracks the elapsed time between locally finishing and the last device finishing, a parameter that the master may query over the serial protocol. A single device on a bus gives a delay of around 3 us (most of which is due to the non-zero execution time of the routines to start a new move), while buses with four devices have longer delays of up to 60 us. Furthermore, the timer serves a secondary purpose, generating a timeout interrupt and putting the axis into an error state if the synchronization delay becomes too large (nominally, 100 ms). During execution, the Master node can totally ignore the sync line, using it only to halt execution (by tying the sync line low and preventing slaves from starting the next move) or start execution (by tying the sync line low while queueing moves, then tri-stating it to start the first move). This isolates the Master node from the real-time aspects of the axis controllers.

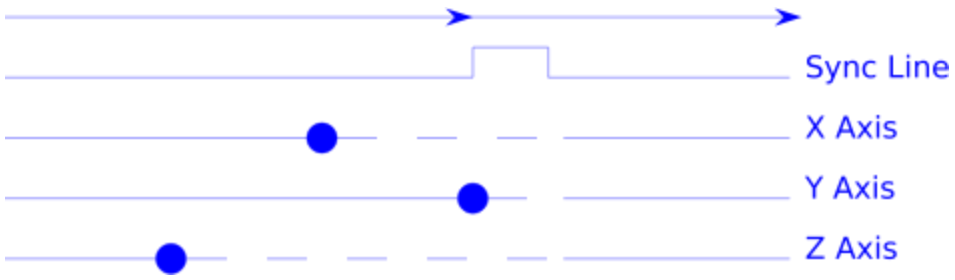

Figure 2. A depiction of a synchronization protocol during a move transaction, showing the state of the sync bus and each axes' sync pin (solid axis line implies ground, dashed implies high-impedance).

RESULTS AND DISCUSSION

For this project, a breadboard-level prototype was constructed, connecting power, ground, and communication lines between the RAMPS motherboard and each of the slave processors. This assembly was then connected to a Makerbot 3D printer (see Figure 3). After a great deal of testing and debugging, full motion of the axes was verified, however a thermistor issue prevented the extruder from heating, so printing was not attempted.

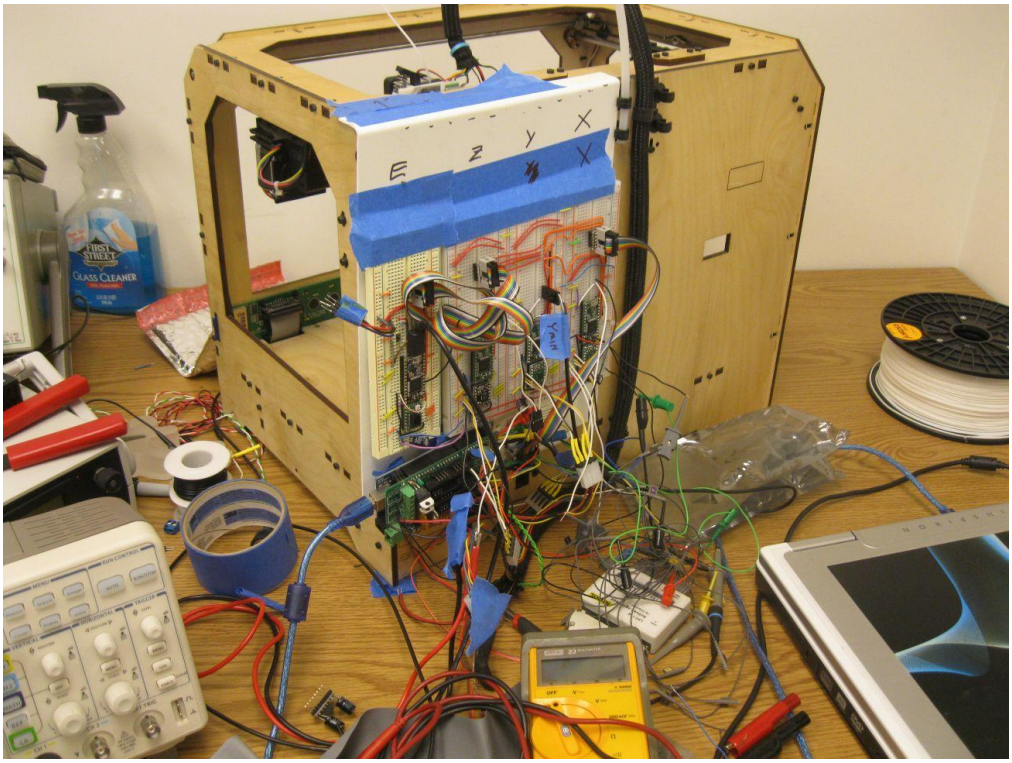

Figure 3. Final Device. The four breadboards in the top row are the axis controllers; a RAMPS 1.2 mainboard is in the lower-left.

The I^2C serial interface was shown to keep pace quite well with the Marlin board's ability to receive and process G-Code over the USB serial link. This means that the new serial bus should rarely be an issue in ensuring printing continues smoothly. Additionally, even with four motor drivers running in close proximity, we experienced almost no problems caused by interference on the serial communications lines.

Curiously, when moves operate at feedrates above about 300 mm/min (or many short moves are executed in quick succession), experimental results indicate some kind of corruption is occurring which prevents motors from getting the correct instructions from the master, resulting in gibberish moves that differ drastically from those intended by the original G-Code. We hypothesize this behavior could be due to interference with the I^2C bus caused by the motor drivers, or by a curious reduction in the ability of the USB serial interface to load new moves.

However, we have demonstrated that this platform works. The improved interrupt prioritization of the K20 MCU and the reduced number of tasks which need to be performed at once enables the step commands issued to be precise and regular. The slave drivers remained synchronized to within 1ms of each other over the course of our testing, indicating that the printer will function in a coordinated manner. The door is now open to extending this platform to allow complex motor control computations, additional axes, and more.

The total cost of implementing the system described in this report is an important consideration. Because each axis now has its own processor, additional costs for those MCUs will be present. In our design, we utilized development boards from PJRC.com, which cost $20 each. The drivers used were the same as originally packaged with the RAMPS kit, so the total cost for this development was roughly $80 ($20 for each of the four axes). Future design revisions could integrate a comparable ARM MCU ($3.50 in quantity 100) and stepper driver into a dedicated PCB, reducing the absolute cost to $15-20 per axis, at the expense of increased design complexity and assembly time. While it may be possible to reduce the

expense of this platform to the $50 mark, actually doing more complex control or running additional axes would again raise this number significantly.

At present, our system requires a full Arduino Mega and RAMPS board ($80-$180, depending on source) to handle extruder temperature regulation and communication with the host PC. However, there is nothing preventing future versions of this system from replacing RAMPS with a dedicated ARM motherboard with breakouts for the heaters, sensors, LCD/SD card, and I2C bus, while using less than half the current RAMPS board area and significantly reducing complexity and cost. Alternately, a low-cost embedded single-board computer such as a Raspberry Pi ($25) could serve as a host computer that directly interfaces with the motor control boards.

CONCLUSIONS AND FUTURE WORK

In the preceding sections, we have described, implemented, and validated a platform and communications architecture that promises to provide a variety of benefits to the open source 3D printing endeavor as printers become progressively more complex and demand greater computational resources. Additionally, splitting the code into modules running on different processors reduces the complexity of each processor's tasks and makes validating the code a more tractable problem, at the expense of sometimes-tricky serial communication issues encountered during initial development. Near full backwards compatibility with the Marlin RAMPS firmware and the Pronterface frontend was maintained, and the robust, tested CNC-driver algorithms from Grbl were implemented on the slave devices. This increases the solution's appeal, in that all of the existing slicing and printer control software can be retained, with significant modifications to Marlin implemented as a fork on GitHub.

The implementation brought the concept to realization in breadboard, and demonstrated its ability to control a 3D printer. Although several areas remain in which further debugging and optimization is needed, this project represents a proof of concept of the design. The total cost

incurred by the additional processors and peripheral electronics was roughly $80, although a more integrated PCB design might reduce this figure.

Going forward, some polishing is still needed in the code base, as well as additional testing. A design revision which integrates all of these components onto PCB's is a logical next step, as well as incorporating the ability for the slaves to use more complex motor control methods. Additionally, a protocol revision allowing for addressing temperature control nodes on the same bus as motor drivers could increase the system's utility at little additional cost. Finally, publishing the designs, codebase, and documentation on the web for others to use and modify will allow others further revise and enhance the system.

Ultimately, this project reinforced our previously-held belief that systems which involve multiple processors working together to perform real-time synchronized tasks, the development task is difficult. Our ultimate failure to produce a system that works at least as well as the original, unmodified RAMPS setup shows just how difficult embedded parallel development is. This project was very valuable, however, in growing our skills as developers, in exposing us to the intricacies of serial and synchronization protocols, and in developing the foundation of a working platform able to improve the performance of 3D printers everywhere...once we squash just a few more bugs.

REFERENCES

[1] RAMPS. http://www.reprap.org/wiki/RepRap

[3] E. Moyer, “A Gestalt Framework for Virtual Machine Control of Automated Tools,” Masters Thesis, MIT, Boston, MA, 2008. http://pygestalt.org/

[2] PJRC Teensy 3.1 Development board. http://www.pjrc.com/store/teensy31.html

[4] Grbl. https://github.com/grbl/grbl

APPENDIX A: COMMUNICATION PROTOCOL

Master-slave communication takes place over a packet-based binary protocol consisting entirely of master-initiated transmissions to a particular slave, followed by a mandatory response. Messages follow a fixed format, starting with a non-null byte that determines the message type. A number of payload bytes then follow, the length and structure of which are determined by the message type. Finally, a checksum byte - the exclusive-or sum of all previous bytes - concludes the message. Responses follow a similar format, starting with a non-null status byte, signaling whether the message was decoded and executed successfully, followed by a number of payload bytes (fixed length depending on message), and then a checksum byte. Communication errors are signaled with a null status byte, and in the case of a master requesting more bytes than a slave expects to reply with, responses are extended with null bytes to the correct length.

Message Types

Initialize

This message resets an axis to a known state, clearing error conditions, emptying all queued moves and releasing the axis' synchronization line. Both the message and response packets contain information on the current hardware and firmware revision numbers of the master and slave.

Status Query

The master uses this message to request information about an axis' current state. The message has no payload data, and the response payload contains a 1-byte status code (ok, mechanical fault, electrical fault, control system fault, or system timeout) and a byte containing

the number of moves currently queued for execution on the axis.

Home Axis

This message initiates an axis' homing routine, the parameters of which are configured via the protocol's general purpose parameter getter/setter mechanism. Upon receiving this message, an axis immediately sends a single status byte response, enters its homing state, and pulls its synchronization line low for the duration of the homing routine.

Queue Move

This message adds a command to an axis' local motion queue. The payload contains the distance and direction the destination axis must move (measured in motor steps), the length of the overall move (also in motor steps), and parameters that define the acceleration profile. If the destination axis has no more room in its queue, a special response status byte is used to indicate this, and the move must be re-sent once there is room for it.

Get and Set Parameters

Device configuration and status reporting is implemented via two message types that allow the master to access and modify a number of 32-bit parameters on the device. The "set" message type contains a single-byte parameter number and a four-byte value for the parameter, while (symmetrically) the "get" message type contains a 1-byte parameter number and receives a 4-byte parameter value in response. Examples of configuration parameters include motor microstepping, homing direction, and limit switch inversion. This mechanism is also used to implement more in-depth error reporting and report synchronization errors.